CMU-CS-80-[illegible]

An Inter-Process Communication Facility for UNIX

February 4, 1980

Revised: 17 March 1980

Richard F. Rashid
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-80-124

A

An Inter-Process Communication Facility for UNIX

February 4, 1980

Revised: 11 June 1980

Richard F. Rashid

Department of Computer Science

Carnegie-Mellon University

Pittsburgh, PA 15213

Abstract

An inter-process communication facility implemented at Carnegie-Mellon University for VAX/UNIX version seven is described. This facility was designed to provided language, operating system and machine independent communication between processes performing distributed computations. Its relationships to previously existing UNIX facilities and other systems for distributed computing are discussed. [illegible]

Keywords: Inter-process commmunication, networking, UNIX, network operating systems, distributed computation.

message processing, compatibility, (KR)

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

1. Introduction

This paper describes an inter-process communication facility which has been implemented at Carnegie-Mellon University for VAX/UNIX Version 7. The ideas on which this facility were based derived primarily from the author's experience with the design and implementation of the RIG system at the University of Rochester. In addition, there have been a number of attempts to produce resource-sharing or message-based UNIX or UNIX-like systems from which the author has liberally appropriated ideas. Notable among these is the TRIX system currently being implemented at MIT.

A good inter-process communication facility is important because of the need to provide access to resources distributed over local and national computer networks in a uniform manner independent of language, network location, or host operating system. Traditional techniques for inter-process communication in UNIX, such as shared files and pipes, are difficult to use for asynchronous message-style communication, and difficult to generalize to a network environment. MPX (multiplexed files), a facility new to UNIX in Version 7, solves many of the problems associated with shared files and pipes, but fails to provide consistent, uniform access to both process- and kernel-supplied resources. In addition, it places its emphasis on a particular kind of communication style, that of file-like data access, to the detriment of more general inter-process cooperation.

The UNIX IPC described here attempts to overcome these problems. It provides a simple mechanism for passing messages between processes on a single UNIX system. Moreover, the design is such that transparent communication between processes on different UNIX systems or even on different host operating systems connected via a network can be implemented without further modification to the UNIX kernel.

Compatibility with older operating system concepts provided by UNIX (e.g. files, devices and pipes) has been stressed in the IPC design. A flexible directory lookup mechanism is provided which allows objects of all types to be uniformly named and referenced, even if they reside on external UNIX systems or on systems running non-UNIX software.

2. Constraints on the design

From the outset, a number of constraints were placed on the design of the IPC facility. They were:

- *Language independence.* The IPC was designed specifically as a tool for interconnecting distributed systems consisting of processes written in many different languages. Individual language systems (such as Ada) might define for their own purposes mechanisms for inter-process communication (e.g., tasks or remote procedure calls) which would layer on top of the IPC facility, but it was considered important that the IPC itself be defined in a language independent fashion. For this reason a loosely-coupled message-passing approach to inter-process communication was adopted.

- *Operating system independence.* The UNIX implementation is but the first in a series of planned implementations of the IPC (see below). It was assumed from the outset that the IPC facility would link together a number of different operating systems running on machines with different process structures and virtual memory support. The net effect of this constraint was the adoption a notion of communication which was inherently local to a given operating system kernel and did not make assumptions about the process structure or name space of a particular host operating system.

- *Machine independence.* The desire to use a single facility for communication between machines with different instruction sets and word sizes and data representations led to adoption of a standard but extensible protocol for the exchange of typed information.

3. Basic Definitions

A UNIX system consists of a *host machine*, a *sytem kernel*, and a collection of *processes*. The function of the system kernel is to provide an execution environment for processes running on its host machine. This includes virtual memory management, the allocation of computational resources, and inter-process communication. It may, in addition, provide other services such as file system access and primitives for manipulating processes (e.g. process creation, suspension and destruction). In this latter role the kernel can be considered a server process much like other server processes discussed in this paper.

Processes are the basic functional units. They perform computations and manage resources. Typically, processes communicate via *messages*. A message is a collection of typed data objects, constructed by a process, passed to the system kernel via a message kernel call and managed by the system until delivery to its final destination. The contents of a message are primarily determined by the sending process, but a message header containing certain system related information may be supplied by the system kernel.

The services and facilities provided by a process are made available to other processes through one or more *ports*. A port is a protected kernel object into which messages may be placed by processes and from which messages may be removed. Ports are intended to be used by processes to represent specific services. For example, a file system process might associate a separate port with each open file, or a virtual terminal handling process might allocate a port to represent each virtual terminal. However, no restriction is placed on their use. Ports may be used by processes to communicate information in any mutually convenient way.

Logically associated with each port is a FIFO queue of finite length on which reside messages which have been sent to that port but which have not yet been removed from it by a process. The ability to remove messages from a port is called *receive access*. Only one process may have receive access to a port at a time. Receive access to a port may be transferred to another process in a message.

Ports cannot be directly manipulated or named by a process. Instead, processes are provided by the kernel with a secure *capability* to send a message to a port and/or extract (receive) a message from it. This capability is a local name for a system object just as a UNIX file descriptor is a local name for a system maintained file, pipe, or device. Port capabilities may be passed in messages, handed down to process children, or destroyed. A given port may have only one local name for a given process at a time. Whenever a port name is passed in a message the system kernel must map that name from the local name space of the sending process into the name space of the receiving process.

The ability to manipulate access to ports allows for the redirection of communication from one process to another and the explicit management of communication between two processes by a third process. It also allows a port to be used to refer to a specific service or process-provided object even in situations in which that service or object is handled by different servers at different times. Neither the address (i.e. its location) or the name of a process can be determined from a capability to send a message to one of its ports.

See Figure 1.

4. Ports

A port is a FIFO queue of messages. Messages may be added to this queue by any process which can refer to the port via a local name (or capability). Messages may be removed from the queue only by the single process with receive access to the port.

4.1. Creating, accessing and destroying ports

A port is created by a process through an *AllocatePort* system call. It is a system object, distinct from the process which created it, but is initially *owned* by the creating process. The result of the *AllocatePort* call is a local port name which refers to the created port. This name is logically an index into a correspondence table maintained by the kernel for each process. It has meaning only when used by that process.

Initially the owner of a port also has receive access to it. Ownership of a port and receive access are, however, logically distinct. Ownership of a port may be passed in a message from one process to another, but not shared. Receive access to a port may also be passed in a message but not shared. These restrictions prevent multiple servers from managing the same queue, and are necessary to avoid serious problems which occur when access to a single queue is shared by processes on different machines.

If a single process both owns and has receive access to a port, that process may destroy the port by performing a *DeallocatePort* system call. A process with a capability (local name) to a port may release the capability via the same system call.

A port is automatically destroyed when its owner and the process with receive access to it both die. In either case, all processes with access to that port are notified via emergency messages (see below). If the same process does not both own and have receive access to a port, then the deallocation of the port name by either process results in an emergency message being sent to the other accompanied by full access rights to that port (i.e., both ownership and receive rights).

The purpose of distinguishing receive access from ownership is to allow a process to take over services or functions provided by other processes in the event those processes should die or malfunction. This is particularly important when writing fail-soft software and can also be used to provide orderly shut-down of services after catastrophic failures.

Port names may be explicitly managed by taking advantage of a use count field provided by the kernel for each local name. Upon creation the use count for a local name is zero. A *DeallocatePort* call on a port name with zero use count releases the name. It is possible, however, for a process to perform an *IncrementUseCount* call with an existing local port name as an argument. This increases by one the use count field for that name. Likewise, a *DecrementUseCount* call is provided to reduce the use count of a port name by one. A *DeallocatePort* on a port name with a non-zero use count has no effect. This feature is designed to allow a process, which may be performing several distinct tasks, to store away a port name without worrying about the possibility that the name could be accidentally deallocated in the course of performing a different task.

4.2. Sending and receiving kernel messages

For the purpose of sending and receiving messages the kernel can be viewed as a process. This allows services which are normally regarded as process-provided to be optionally performed by the system kernel to increase efficiency. An example of a function which is logically process-provided but can be (and normally is) performed by the system kernel is the UNIX file system.

4.3. Queueing messages for a port

A message is sent from a process to a port. Although the kernel may append to the message header information describing the sending process (e.g., for the purpose of debugging), messages are essentially anonymous. All messages sent by the same process to a particular port are guaranteed to arrive in the order in which they were sent. There is no other guaranteed order of message events (see below).

Messages to a port are normally queued in FIFO order. However, an *emergency message* can be sent to a port and receives special treatment with regard to queueing, flow control, and the generation of signals or software interrupts (see below).

4.4. Flow control

The message queues attached to ports have a finite length. This prevents a sending process from queueing more messages to a receiving process than can be absorbed by the system and provides a means for controlling the flow of data between processes of mismatched processing speed.

Subject to implementation restrictions on maximum port size, the process owning a port is allowed to specify its *backlog* -- the maximum number of messages which may be queued for that port at one time. Should a process attempt to send a message to a full port, one of three things may happen depending on options specified by the sender:

1. The process is suspended until the message can be placed in the queue.
2. The process is notified of an error condition (after perhaps allowing itself to be suspended for up to a specific period of time wiating for the message to be sent).
3. The message is accepted and the kernel sends a message to the sending process when that message can actually be placed in the queue. A maximum of one message per sending process per receiving port may be outstanding in this fashion.

These three options correspond to three different programming situations:

1. The first option is the one most likely to be used by a 'user process' when communicating with a 'server process'. In this situation the user process does not care if it is suspended for some time waiting for a message to be delivered to the server (as in the case of a remote procedure call).
2. The second option is used when a process does not care whether a particular message is sent to a destination, but is using the message only to wake up a dormant partner. The fact that other messages are in the queue for the partner's port indicates that the partner is already scheduled to be activated.
3. The third option is the one most likely to be used by a service process when dealing with a user process. The server probably cannot afford to be suspended waiting on a user to clear its queue. It may also not want to just throw away the message or poll the user explicitly until the message can be sent. The system provides an explicit message event corresponding to the unblocking of the user's port queue.

The flow control scheme described here is not the only reasonable technique for restricting the flow of messages between processes. Other possible mechanisms include limiting the total number of messages which may be outstanding from a given process (as in Brinch Hansen's RC4000 system or CMU's Hydra) or limiting the outstanding messages from a given process to a particular port.

Experience with RIG, however, has shown that the exact flow control policy is much less important to overall system performance than the higher level communication protocols devised to solve individual problems. Even more important to the functioning of the system is the provision for events to indicate when ports become unblocked. This prevents needless and expensive polling in server processes.

5. Messages

5.1. Message structure

What structure, if any, should be imposed upon messages is a sensitive issue. Ideally, a message should be considered a collection of program objects the structure of which is preserved by transmission from one process address space to another. This goal is difficult to attain, however, because the system kernel cannot make assumptions about the allocation and deallocation of free storage in a process's address space.

The use of completely unstructured messages which are then interpreted as needed by user processes also has problems. Some parts of a message (e.g., local port names) must be interpreted by the kernel because they require translation to be meaningful to another process. Moreover, in a heterogenous network where processors of different word sizes and storage conventions are communicating, only typed information allows the transparent transfer of data items such as integers, reals, and strings. Different rules for byte packing (as, for example, on a PDP-10 and a VAX-11/780) make even simple byte streams difficult to generalize to a heterogenous network without some knowledge of the information contained in the byte stream.

Efficiency considerations also favor structured messages. Most information communicated between processes is structured in that it represents data items of different types. The use of unstructured messages for such data is can be expensive because the packing and unpacking of structured messages into an unstructured linear form adds a layer of overhead which, although not strictly kernel overhead, serves to increase the cost of communication. In a heavily message oriented system such as RIG, this has been shown to add as much as a factor of two to communication costs.

In defining the IPC, the decision has been made to provide a message representation which has structure while at the same time preserving the ability to send and receive raw data. Messages are defined by a header which contains a descriptor for the message data to be sent (see Figure 2). A descriptor is of the form *<Type,NumberOfElements,Data>*.

Type can either be system defined or user defined. System defined types are:

- TypeUnstructuredData

- TypePortCapability
- TypePortOwnership
- TypePortReceiveRights
- TypePortAllRights
- TypeDataDescriptor
- TypeLinearStructure

NumberOfElements is the number of objects in the data portion of the descriptor and the nature of the data portion is defined by its *Type*. The data portion of a descriptor may contain more than one object of the kind specified by the descriptor's *Type*. In particular, if the *Type* is TypeUnstructuredData then *NumberOfElements* is the number of bits of unstructured data. If the *Type* is TypeDataDescriptor then the message structure represents a tree, where the nonterminal nodes are data descriptors and the terminal nodes represent the actual data.

The specification of a user type includes the bit size and packing characteristics of its elements. Although not logically necessary, certain user type codes will have meanings understood by network servers and programming language runtimes and will therefore be redefined at the peril of the user. The list of such reserved user types may vary for particular installations. An example of possible reserved types would be:

- TypeInteger(8,16,32,36)
- TypeReal(16,32,48,64)
- TypeUnsignedInteger(8,16,32)
- TypeCharacter(8)
- TypeBoolean(1)

A data descriptor may reference the actual message data in one of three ways:

1. The data may be stored in the same block of storage as the descriptor, immediately following the *NumberOfElements*. (Figures 2 and 3)
2. The data is in a block pointed to by the word immediately following the *NumberOfElements*. (Figures 3 and 4)
3. Immediately following the *NumberOfElements* is an array of pointers (*NumberOfElements* long), each of which points to a separate data element.

The ability to manipulate data via either pointers or direct reference allows data to be gathered from a numbe^r of different sources or constructed in a single buffer. For those applications with a need for greater structuring, a tree structure may be constructed using either system or user defined types. System defined types, in particular local port names, can be converted by the kernel as necessary.

The internal form of a message is composed of three parts: the header information and linearized forms of the structure information and the raw data. When receiving a message a process may use the *Type* field of the message header to specify the form desired for the received data. By specifying the type LinearStructure, a process requests to explicitly receive the structured and unstructured information content of a message in two preallocated blocks of storage. Prior to performing the receive, the process must specify the sizes and pointers to the two blocks in the data portion of the message.

By specifying other types, the receiving process may direct information directly into specific locations in its address space. For example, a file system may send a 4-word header together with a disk page of data as a single block of storage. The receiving process, on the other hand, may specify that the 4-word header and data are to be stored into separate areas of storage. The use of identical message structures for sending and receiving a message result in the complete preservation of structure across a message transmission. An attempt to receive a message into an incompatible message structure results in an error.

One effect of this scheme is to allow simple unstructured information to be sent with a minimum of overhead. Thus processes which wish to be unaware of message structuring need not pay a price for its support by the kernel.

5.2. The message header

The message header contains a small amount of system required message information and is used to form an anchor for the structured part of a message. At minimum it contains a capability for the destination port of a message and a field for a capability to be used for a reply (which may be empty). The destination and reply ports will more commonly be referred to as the *remote* and *local* ports, respectively. The header also contains an ID field to be used for descriminating messages and the *Type*, *NumberOfElements* and *Data* of the message.

The header may be checked without actually receiving the data portion of a message by calling *Preview* (see below). The purpose of this facility is to allow a process to check on the ID of the message received and select an appropriate message structure for receiving it.

5.3. Message types

The type of a message contains information which determines the kind of service it requires of the IPC. The following service classes are provided:

- *Flow control*

  Normally messages are flow controlled according to the above procedure. The message type can specify, however, that a larger flow control backlog should apply to this message. This allows special high priority messages to be sent to otherwise full ports.

- *Priority*

  A range of message priorities is provided. Messages waiting at the same port with different priorities will be received according to the order of their priorities.

- *Sequentiality*

  Messages from the same process to a particular port with the same priority are normally guaranteed to arrive in the order they were sent (FIFO). It is possible to relax this constraint by noting in the message type that the order of reception of a particular message is unimportant.

- *Reliability*

  Messages are guaranteed to arrive at their destination reliably, as long as the destination exists long enough for the message to be received. A message could simply be advisory, however, and it may be unimportant that it arrive at its destination. A message may therefore contain in its message type an indication that it may be delivered unreliably.

- *Maximum age*

  A message could become out of date after a certain length of time has passed. A message may therefore be marked as having a maximum age after which it may be destroyed rather than delivered.

- *Security*

  A message may be marked as requiring special procedures for ensuring the security of its data. Messages containing password information would fall into this class. Messages marked as 'secure' are encrypted when transmitted on a network or placed on a storage medium which may not be secure.

Two message types have been given special names and play a dominate role in communication:

1. *Normal messages*

   A 'normal' message is flow controlled, sequential, reliable, not secure, of lowest priority and has no maximum age. This is the default message type and is assumed to satisfy most communication requirements.

2. *Emergency messages*

Emergency messages are specially flow controlled, sequential, reliable, not secure, of highest priority, and no maximun age. Emergency messages play an important role in error handling. Because of their high priority, they are guaranteed to be received before any normal messages sent to a port. Their purpose is to allow urgent information to be delivered to a process regardless of that process' current message backlog or message queue. They are used for error notification, special event processing, and debugging.

The notion of message type allows the programmer the ability to specify the exact requirements of his IPC use. This gives the underlying system more information about a message and thus makes possible optimizations in the delivery and management of messages.

The service classes associated with sequentiality, reliability, maximum age, and security are considered advisory. The system may choose, for example, to deliver a message reliably which has been marked as unreliable without affecting the correctness of the programs which use unreliable messages. For the most part they have a direct effect only on the management of message traffic by network servers and other intermediary processes. Priority and flow control, however, may be important to program correctness and cannot be changed without possibly introducing deadlocks.

5.4. Waiting for messages

Although particular protocols may specify sychronous behaviour, the receipt of a message is inherently an asynchronous event. In a network environment in particular it becomes possible for error conditions to cause messages to be sent to a process at almost any time. Mechanisms must therefore be provided for a process to to check the state of its ports, to wait for activity on one or more of its ports, and to receive messages selectively. It should also be possible for a process to specify a maximum amount of time to wait for a message before reawakening (see below).

Four basic primitives are provided for accomplishing this task. *Receive(SetOfPorts,Message,Timeout)* waits for at most *Timeout* milliseconds and if a message is available during that time from any of the ports designated by *SetOfPorts* then the message is read into *Message* and a boolean value of true is returned. *MessageWait(SetOfPorts,Timeout)* performs a similar function, but does not receive the pending message. It returns the capability of the port from which the next message would be received. *Preview(SetOfPorts,Message,Timeout)* is again similar to *Receive*, but it reads only the header of the next waiting message and does not actually dequeue that message from the port queue. *PortsWithMessagesWaiting(SetOfPorts)* is an informational routine which checks the status of all ports and returns the set of ports with messages waiting.

5.5. Messages and signals

In the case where a process does not wish to wait for messages explicitly, it is possible to enable a signal (i.e. software interrupt) which will be triggered upon message reception. The signal service routine can then receive the incomming message and process it or notify the main program of the event in a user defined manner. A mechanism such as this can allow processes which are inherently compute bound to react quickly to incoming messages without using some form of message polling.

A separate signal is used to indicate the arrival of emergency messages and trigger error recovery procedures.

5.6. Exceptional condition handling

Distributed programming imposes a heavy responsibility to handle a multitude of error conditions. Message activity can be pipelined or multiplexed, and the relationships between incoming and outgoing messages can be much richer than in a conventional programming environment (i.e., one in which subroutines are used as the primary structuring mechanism). A faulty process could conceivably crash many processes by sending illegal messages, making it very hard to identify the source of the problem. As a practical matter, it is difficult to ensure complete compatibility between similar programs written by different individuals in different languages. Supposedly interchangable processes may differ in subtle ways. A failure in a message-based system can quickly lead to finger-pointing.

A variety of facilities for protection, error detection and error handling have been built into the IPC. Illegal process addresses are noted within the send and receive primitives, and the appropriate error returned to the offending process. In addition, processes are protected from accidental or malicious access through the use of port capabilities rather than global port or process identifiers. Whenever a port is destroyed the kernel notifies all processes which still have access to that port via an emergency message. Emergency messages themselves are important because the give processes a a way to reliably communicate errors in situations where normal communication channels are blocked. Software interrupts allow processes to handle error conditions without interferring with normal execution.

6. Compatibility with UNIX

In the previous sections the IPC facility has been defined in a manner largely independent of UNIX. This is consistent with the design criterion that the underlying facility should be operating system independent. In the UNIX implementation of the IPC, however, code has been written to allow a mapping between facilities which can be made available by processes through the IPC and existing kernel functions.

6.1. Uniform Reference: Ports, Pipes, Files, Devices

A number of services performed by the UNIX kernel, such as file and device input/output, are no different in kind than similar services which could be provided by a UNIX process and made available through messages. The act of performing a *Read*, *Write*, *LSeek*, or *IOControl* system call can be equated with the act of sending of a ReadMessage, WriteMessage, SeekMessage or IOControlMessage to a port and waiting for a reply on a different port belonging to the requester. The kernel can thus be viewed as a process in its own right providing services through messages. A file descriptor for a kernel service such as an open file acts analogously to a pair of ports, one local to the requesting process and one owned by the kernel, which can be used to send and receive messages conforming to a standard protocol understood by the kernel.

This correspondence between kernel calls on file descriptors and message interactions using a pair of ports provides the fundamental link between the older UNIX notions of files, devices and pipes and the UNIX IPC facility. A file descriptor may be constructed from a <local,remote> pair of port capabilities and all the normal UNIX system calls relating to file descriptors may be applied to it. Whenever a UNIX call such as *Read* or *Write* is applied to a descriptor constructed in this fashion, a message is sent to the remote port describing the request and the requesting process then waits for a reply on its local port.

Complete uniformity of reference is provided by allowing 'real' file descriptors (i.e., file descriptors provided by the kernel for kernel maintained objects) to act like a <local,remote> pair of ports to which messages can be sent and from which message can be received. Thus a ReadMessage can be sent using a file descriptor as a remote port argument even if that file descriptor describes a kernel maintained file. A process need not know whether a particular service referenced by a file descriptor is implemented by the kernel directly or by a process. Use of incorrect message protocol when dealing with kernel objects results in an error message in just the way that an error message would be sent by a process providing the service.

The ability to view system kernel defined objects as accessible via messages provides a syntax through which asynchronous file and device input/output can be provided. Assuming that kernel device drivers are modified to allow asynchronous I/O, a process can initiate a request by sending a ReadMessage or WriteMessage using an appropriate file descriptor and then at some later time receive the reply. An important reason for adopting messages as the fundamental unit of communication is the ease with which it allows the specification and handling of asychrony.

6.2. Directory Lookup

A capability to a port may be obtained through the standard file lookup mechanisms of UNIX and access to it is governed by the UNIX file access control mechanisms. An *AssertName* kernel call allows a process to associate a name in a directory with a capability for one of its ports. This correspondence exists until that capability is destroyed (either explicitly or by process death) or until it is removed explicity by that creating process (using a *RemoveName* call).

This facility may be used directly as a name lookup mechanism for message communication through the use of the *Lookup* call which takes as its argument a string name and returns the corresponding port.In addition, name loookup can serve to provide a means for processes to make file-like or device-like services accessible through the UNIX *Open* system call. In standard UNIX systems, whenever a UNIX process issues an *Open* the UNIX file directory is searched. If that search is successful, a file descriptor is returned which may refer either to an actual file, a multiplex file channel, or a device. With the addition of the IPC facility, a file descriptor corresponding to a <local,remote> pair of ports may also be returned by the *Open* request. If the pathname specified by the *Open* contains the name of a special capability file then further processing of the search is halted, a local port is created for receiving a reply, and an OpenMessage containing the unparsed pathname and local capability is sent to the port described by the special file. If the process to which this port belongs wishes to honor the request, it replies with an OpenReply message containing a port capability to use for further communication. This capability is matched with the local port to form a file descriptor which is then returned as the result of the *Open*. See Figure 5.

The way in which the matching capability for an *Open* is obtained is completely up to the process which intercepts the request. The pathname requested by the calling process is supplied as part of the OpenMessage. The process receiving the OpenMessage may chose to parse and use this information, or it may chose not to handle the request but instead to pass it on to another process for final resolution.

6.3. *Fork* and other things

The UNIX call *Fork* is the only way new processes can be created. The definition of *Fork* is such that all files open before a *Fork* will be shared by both parent and child processes after it. The IPC facility preserves the semantics of the call. For file descriptors which are <local,remote> pairs of ports the child process is provided with a copy of the remote port capability and a new port owned by the child is created to serve as the local port. If the remote port capability of the file descriptor belongs to the parent, only the right to send messages to the remote port is transferred (not ownership!).

The newly created process can use these file descriptors (which of course have the same values as they had for the parent) in the same way the parent uses its version of them. The protocol for reading, writing, etc. objects referred to by such file descriptors requires that all messages use the local port of the file descriptor as their own local port. Thus should the child process perform a *Read* or *Write* the request message would be sent to the same port as for the parent and the reply would be sent back to the child. The semantics are exactly equivalent to those for shared files.

No port capabilities other than those used in file descriptors are transferred from parent to child. However, the child is given capabilities to certain ports which can handle for it such things as name lookup and file access. These ports may belong to a system kernel (possibly on a remote machine!) or may belong to processes.

Whenever a new process is created, two ports, one owned by the new process and one owned by the system kernel, are also created. These ports are the made accessible to the parent process via a *ChildPorts* call. The first of these is intended to be used for initial communication to the child process by the parent. The latter is a control port designed to receive messages requesting termination, suspension or status information. The control port belongs to the kernel (viewed as a process) just as UNIX files are really represented as ports belonging to the kernel.

7. Networking

Given the UNIX IPC outlined in the previous sections, it is possible to describe the implementation of a transparent network IPC which does not require further kernel modification. In fact, the kernel requires no knowledge of networking or networks for such a mechanism to be provided.

Assume the existence of a network server process running under UNIX and providing reliable, flow controlled communication paths to other machines on a network. A communication path for messages to a port on another machine on the network may be identified with a port which belongs the network server and to which processes may send message to be forwarded across the network. In other words, a process A on machine X can possess a capability for a port to which messages can be sent destined for a process B on machine Y. This capability is in fact a capability for a port owned by a network server on X, but this is both unknowable and unimportant to A. The exact nature of the connection, the protocols used, the topology of the network, or even the existence of multiple competing networks linking X and Y is hidden from process A. It would even be possible for B to migrate from one machine to another on the network without process A being aware of the change as long as the network servers involved could handle the transfer of the logical communication channel between A and B. See Figure 6.

One requirement of this scheme is that there be a way for network servers to know when a message contains a port capability. Along with each such capability sent in a message goes an implicit network connection which would be needed should a message be sent to that port. It would therefore be the responsibility of the network server to convert any references to capabilities found in messages forwarded over a network into capabilities local to its host machine.

There is no requirement, however, that there should be only a single network server or that all network servers should use the same communication protocols or even that 'network servers' actually implement communication on a physical communication link. This allows multiple competing versions of network software to be written and debugged or multiple network links to be provided without affecting the system kernel or existing software.

Where efficiency is critical, the fact that the kernel can act like a process allows the implementation of the network server in the UNIX kernel.

Message structuring is important not only for handling the transmission of port capabilities across networks but also for data conversion in networks with machines which have widely varing data representations for integers, reals, characters, etc. As stated above, it would be appropriate in such networks to reserve certain 'user types' which would to have network understood meanings. In this way network servers could provide for data conversion across machine boundaries. Specifically, process A on machine X is communicating with process B on machine Y. A sends B a message containing arbitrary data. The kernel on X packages the data into a host-dependent linearized form and delivers it to the network server, which then delivers the data across the network in a manner which Y can understand. The network server on Y unpacks the data into the host-dependent linearized form for Y and delivers the message to its kernel. If word-size mismatches may occur, B can specify in its receive that the data should either be truncated or expanded.

8. An Example: A Simple Network File System

As example of the power of expression provided by UNIX IPC, a simple network file system for UNIX can be described. Assume that each UNIX host machine on a network has a running remote file server process. The function of this process would be to provide kernel-like access to files via *Open*, *Read*, *Write*, *LSeek*, etc. messages. Assume also the existence of network server processes providing network IPC as described in the previous section. The ability to perform *Open* requests on files on another machine is then provided by simply placing a capability for the remote file system process for that machine in the directory structure of the local host (through the auspices of the network server). From that point on, all file opens which use the name of the root node of that file system are trapped and sent across the network to the appropriate remote file server. File access provided in this way need be no different than file access provided by the local host's kernel. Access may even be provided directly to remote devices.

9. Discussion

Ports simplify the problems of network communication and protection by decoupling networking from intra-machine communication and providing an abstraction of function which does not depend upon a particular implementation of processes. A port can be used to represent a function independent of how that function is implemented. They can be used to specify kernel objects (such as files or process management functions) or objects maintained by processes. All access to the outside world is mediated through ports with the notion of system call reduced to the notion of sending and receiving messages.

Protection derives from the fact that ports are protected kernel objects to which processes only have a local reference. The right to send a message to a port cannot be forged or accidentally created. This prevents processes which are either buggy or malevolent from gaining fraudulent access to resources and it allows the writer of a process to make a positive statement about the correctness of his program based on precise knowledge of which other processes are allowed to communicate with it.

The existence of access lists also provides the kernel with complete knowledge of the communication rights of processes. It is not possible for a process to hide a reference to another process's port in its private address space. The system can know who is talking with whom and notify processes which have access to a port when that port is destroyed. Knowledge of inter-process communication paths can also be used as an aid in intelligently scheduling systems with tightly coupled communicating processes.

Moreover, the location of a port can change at will as can the nature of the process serving it, without affecting the processes using the port. This feature is critical for process migration and fail-soft performance.

10. Comparisons with other systems

The IPC facility described here can be compared with other facilities for distributed computing constructed or proposed during the last decade.

10.1. Comparisons with RIG

The IPC facility is a lineal descendent of the message-based communication mechanisms provided in RIG[1],[2]. RIG represented a substantial effort in the design and implementation of a network oriented operating system. As of September, 1979 some 90,000 lines of code had been written for that system including facilities for network file access, a virtual terminal facility[3] which allowed a single display to be used to monitor simultaneously executing tasks on a number of different machines (both local and on the Arpanet) and a sophisticated process management facility.

The IPC facility builds upon the accumulated experience represented by RIG. It incorporates the best features of the RIG system and at the same time attempts to provided solutions to some of the many problems encountered in the design and implementation of distributed software in RIG.

Specifically, the IPC differs from the RIG notion of message-passing in that:

- *The notion of process as part of a communication address has been dropped.* In RIG, a message was addressed to a port identified explicitly by the process which owned it. Each process could have up to 255 ports and a port name consisted of a process/port pair. This identification of port with process prevented the transfer of a service from one process to another. This restriction impacted on a programmer's ability to hide the process structure of an implemention of a service from the user of that service.

- *No global name space.* In RIG, the space of process and port names was global across all machines in the system. Possession of a process/port name was all that was required for communication with that process. Process/port names could be stored away in the address space of a user process without the knowledge of the underlying system or even fabricated with good or malicious intent.

  In the CMU IPC facility a process accesses a port through a local name which has meaning only in the context of that process. The conversion from local to system port name is performed by the operating system kernel through the use of a protected access list. Conversion of a system port name to a network connection is performed by a network server. A port name can be placed in the name space of a process either through receipt of a message containing the right to access that name or through the use of a string name and an explicit name lookup mechansism (which would provided the same kind of protection accorded to file name lookup).

  Their are two advantages to this approach. The first is that port names cannot be fabricated, simplifying considerably the task of writing 'correct' distributed programs. The second is that information about the communications structure of the system is explicitly represented in the system (in the kernel access lists) and the network server (in the connection data structures). This allows the system to notify dependent processes when error conditions (such a process death) are detected.

- *Structured messages.* RIG incorporated a primitive mechanism for typing the contents of messages. As long as the only machines on the network had the same word size, instruction set, and data representation this mechanism sufficed. The introduction of the PDP-10 to the message-passing environment presented considerable problems, however.

  A standard protocol for data structuring and typing is part of the IPC facility. Just as it is important that information about the communication structure of the system be represented explicitly to allow for error handling and recovery, it is also important that the type of data in a message be explicitly defined to allow for transparent conversion of data types between processes written in languages with different conventions for the storage of data (e.g. Lisp and C) or between processes on different machines (e.g. PDP-10 and VAX).

10.2. Comparisions with previous systems

Several other systems provide facilities similar to some of those provided by the IPC. Name service, or generic addressing, is provided by DCS[4], DCN[5], and MSG[6]. The Distributed Processing System[7] and recent work by Watson and Fletcher[8] supports the use of communication protocols similar to full hand-shake connections (i.e., procedure calls). DEMOS[9], Roscoe[10], and Thoth[11] are examples of systems built entirely on the use of processes communicating via messages.

The IPC is somewhat unique in incorporating all of these facilities in a relatively coherent manner. MSG, for example, provides three primitives modes of inter-process communications -- messages, connections, and alarms -- together with asynchronous signaling mechanisms and the ability to dictate the sequencing of messages. IPC provides a more uniform message interface: Emergency messages are software interrupts; connections are built on top of messages; sequencing is guaranteed by the reliable transmission protocols.

DEMOS and Thoth rely heavily on messages for process synchronization and communication. Both systems, however, use explicitly shared memory as the primary tool for communicating blocks of storage. Thoth messages, for example, are limited to eight sixteen bit words.

The IPC and Thoth also differ in the style of access to basic system resources. Thoth distinguishes between access to its file system (through kernel calls) and inter-process communication.

10.3. Comparisons with new or proposed systems

10.3.1. TRIX

TRIX[12] represents another system in the general flavor of RIG, Thoth, and DEMOS. The concept of a data stream is reminiscient of a RIG port with the added notion of that a stream descriptor is a capability. TRIX streams are considered full-duplex channels, with messages passing in both directions, but are inherently asymmetric. Requests flow from a requester to a handler and replies are returned on the same stream. Messages are of fixed length, but a request contains a pointer to a buffer in the requester's address space into which data can be placed by the receiver of the message. Message data is not typed.

One way in which the IPC facility differs from the communication system of TRIX is in the use of messages containing typed data. This allows, for example, port names to be passed in the data portion of messages rather than only in fixed message headers. It also allows the system to perform transparent data type conversion between heterogeneous machines.

Another difference is in the asymmetry of TRIX streams. In the IPC ports are simply queues for messages which may have only one receiver but many senders. In TRIX, a stream represents m + 1 queues where m is the number of sending processes. Replies are inherently matched to requests and no provision is made for third party handling of messages (e.g. process A sends a request to B which forwards the request to C which responds directly to A). TRIX has bound relationships between processes to a request/reply paradigm that represents only one of many possible process-to-process relationships. The IPC facility allows a more flexible graph structured relationship between processes. (See[13] for examples of this kind of distributed programming.)

Current TRIX documentation does not specify any mechanism for flow control or error handling. Both are specified by IPC.

Much of the emphasis of TRIX is on the development of a distributed UNIX-like system environment rather than the construction of distributed programming tasks by users. The emphasis in the IPC is on providing a general purpose facility for distributed programming of tasks as diverse as the SPICE system development[14] and AI tasks such as the implementation distributed sensor nets.

10.3.2. Liskov's primitives for distributed computing

Barbara Liskov has proposed a set of message communication primitives which she plans to incorporate into the CLU programming system[15]. A driving force in the specification of those primitives has been the desire to provide a mechanism for automatic recovery. The result has been the decision to define a message passing system which does not guarantee the order of messages and which allows a given message to be delivered at least once but possibly any number of times to its destination. It is the burden of the programmer to define his make his messages idempotent, i.e. capable of repeated delivery without adverse affect. A message system defined in this way has the advantage that the underlying operating system can restart any process from its last checkpoint.

The problem with this approach is its reliance on the notion that no error signal should occur in the event of a failure. Because no error condition is generated in the event of a system crash, messages from restarted processes cannot be distinguished as belonging to a new or previously performed operation. Thus the actions specified by these messages must be idempotent Machine-to-machine communications protocols often have the property that a given message, repeated any number of times, cannot be misinterpreted. Liskov seems to be saying that process-to-process communication must also be of this form to guarantee end-to-end reliablity.

Unfortunately, network protocols are notoriously hard to correctly specify. A logical assumption is that a requirement for idempotency in the specification of a programming task will dramatically increase the time required to write distributed programs. Higher level mechanisms would have to be built on top of Liskov's primitives in order to make them usable.

In contrast, the IPC message system provides a reasonable model for communication under the assumption of no error and guarantees notification of error conditions (such as process death) as long as a communication dependencies exists between two processes. This allows for a programming style to be adopted which does not assume an error will occur which is then augmented by mechanisms for asynchronously detecting and recovering from errors. In addition, it removes one of the two reasons for incorporating timeouts in distributed programs: a programmer need only use timeouts to ensure minimum performance (if desired) not to detect the failure of his cooperating partners.

11. Current state of the IPC

An implementation of the facility currently exists for VAX/UNIX and one is planned for the Three Rivers Corporation PERQ. An implementation is possible for TOPS-10 and TOPS-20 and it appears likely that a VAX/VMS implementation will take place. Work is proceeding on the development of network and file services built around the IPC.

12. Appendix: Structures and Calls

The following describes the set of subroutine calls and structure declarations for the IPC. The purpose of this appendix is explanatory. It does not conform to either the actual names of routines currently being used (primarily because of name length restrictions in C) or to actual structures. Full documentation for the IPC facility as implemented can be found in the appropriate 'Man' files on the various CMU VAX/UNIX systems. The emphasis here is on the kind of functions which are be provided and roughly how they might look to a programmer.

12.1. Structures

```
/* Basic type declarations */

typedef int             LocalPortName;
typedef int             MessageType;
typedef int             PortType;
typedef int             MessageIdentifier;
typedef int             MessageDataType;
typedef int             MessageElementSize;
typedef int             DescriptorPointer;
typedef char            MessageData;

struct SetOfPorts
  {
    Bits[(MAXBITS+31)/32]
  }
```

```
/*
  The implementation of SetOfPorts is as
a bitmap of known length with each bit representing
a portname.  It is desirable for speed and ease
of implementation to provide calls which assume all
ports are referred to or which specify only a single
port in the calls below which take a SetOfPorts as
one of their arguments. If the SetOfPorts argument
is -1 or  a small integer, either all ports or a
specific port is referred to. A pointer-valued
SetOfPorts argument is assumed to refer to a bitmap
structure.

*/

/* Types of messages */

#define NormalMessage
#define EmergencyMessage

/* Message data types: */

#define TypeUnstructuredData
#define TypePort
#define TypePortOwnership
#define TypePortReceiveRights
#Define TypePortAllRights
#define TypeDataDescriptor
#define TypeLinearStructure

/* Type qualifiers: */

#define Inline
#define ArrayOfPointers
#define PointerToArray

/* Send options */

#define Wait
#define ReturnWithError
#define Notify

/* System parameter */

#define MaximumMessageDataSize
#define MaximumNumberOfPointers

/* Minimal message structure */
struct MessageHeader
  {
    MessageType          MsgType;
    PortType             LocalPortType;
    LocalPortName       LocalPort;
    PortType            RemotePortType;
    LocalPortName       RemotePort;
    MessageIdentifier   ID;
    /* Followed by a message descriptor:
       One of
          DescriptorWithPointer
          DescriptorWithoutPointer
          DescriptorWithArrayOfPointers
    */
  };

struct DescriptorWithPointer
  {
    MessageDataType      Type;
    MessageElementSize   NumberOfElements;
    MessageData         *Pointer;
  };

/* Note, C does not have variable length arrays.
   These structures are suggestive rather than
   literally true for any given message.
 */

struct DescriptorWithoutPointer
  {
    MessageDataType      Type;
    MessageElementSize   NumberOfElements;
    MessageData          Data[MaximumMessageDataSize];
  };

struct DescriptorWithArrayOfPointers
  {
    MessageDataType      Type;
    MessageElementSize   NumberOfElements;
    DescriptorPointer    Pointer[MaximumNumberOfPointers];
  }
```

12.2. Calls

```
boolean Send(Message,Timeout,Options)
  struct MessageHeader *Message;

/*
    Send a message. Call returns true if
  it succeeded, false if it was not able
  to send the message. Timeout in milliseconds.
*/
```

```
boolean Receive(Ports,Message,Timeout)
  SetOfPorts *Ports;
  struct MessageHeader *Message;
  Time Timeout;

/*
    If a message is waiting on any of the
  ports specified or if a message comes
  for one of those ports before a specified
  amount of time (in milliseconds) expires,
  then receive that message into the structure
  pointed to by Message and return true.
  Otherwise return false.
*/


boolean Preview(Ports,Message,Timeout)
  SetOfPorts *Ports;
  struct MessageHeader *Message;
  Time Timeout;

/*
    If a message is waiting on any of the
  ports specified or if a message comes
  for one of those ports before a specified
  amount of time (in milliseconds) expires,
  then copy that message's header into the
  structure Message and return true.  Note that
  linear structure form of message is
  returned in preview (but no data is copied).
  Otherwise return false.  This does not
  dequeue the message.
*/


LocalPortName MessageWait(Ports,Timeout)
  SetOfPorts *Ports;
  Time Timeout;

/*
    If a message is waiting on any of the
  ports specified or if a message comes
  for one of those ports before a specified
  amount of time (in milliseconds) expires,
  then return a capability to the receiving
  port. Otherwise return nothing.
  This does not dequeue any message.
*/
```

```
PortsWithMessagesWaiting(Ports)
  SetOfPorts *Ports;

/*
    If a message is waiting on any of the
  ports specified or if a message comes
  for one of those ports before a specified
  amount of time (in milliseconds) expires,
  then return the set of ports with messages
  waiting. Otherwise return the empty set.
  This does not dequeue any message.
*/


boolean AssertName(Name,Port,Access,NameType)
  char *Name;
  LocalPortName Port;
  int Access;
  int NameType;

/*
    Define a correspondence between a string
  name and a port.  The calling process must
  either own or have receive rights to the
  port.  Returns true if the name has been
  entered, false if the name already existed
  or the port name passed in the call was
  somehow illegal.  If NameType is DIRNAME
  then a Locate (see below) when it encounters
  Name as a directory prefix of the name it
  is looking up, will cause a LocateMsg to
  be sent to the specified port and will wait
  for a reply.  Only other value for NameType
  is NORMALNAME.
*/

LocalPortName Locate(Name)
  char *Name;

/*
    Returns the port name associated with a
  string. The access field of the name entry
  is check as for file name lookup.  Returns
  NULL if no correspondence or no access.
*/


boolean RemoveName(Name)
```

```
  char *Name;

/*
    Destroy the correspondence between a string
  and a port.  This call will return true if the
  name exists and the calling process owned or
  had receive rights for the port it referred to.
*/


LocalPortName AllocatePort(Backlog)
  int Backlog;

/*
    Create a port.  A Backlog of zero implies
   the use of the system default backlog.  Negative
   backlogs are ignored.
*/


DeallocatePort(Port)
  LocalPortName Port;

/*
    If the caller is the owner of the port and has
  receive rights to it, destroy the port. If the caller
  has receive rights only, then send a message to the
  owner containing those rights.  If the caller has ownership
  rights only then send a message to the receiver having
  receive rights. If the caller is not the owner or receiver,
  release its capability to the port.  Noop if use count is
  non-zero.
*/


IncrementUseCount(Port)
  LocalPortName Port;

/*
    Increment use count field for port name.
*/


DecrementUseCount(Port)
  LocalPortName Port;

/*
    Decrement use count field for port name.
```

```
*/


ChildPorts(ChildPort,KernelPort)
  LocalPortName *ChildPort;
  LocalPortName *KernelPort;

/*
    Returns in its arguments the data and
    control ports for the last child
    spawned by this process.
*/

DefineSignal(MsgType,Signal)

/*
    In the case where a process does not wish to wait
  for messages explicitly, it is possible to enable a
  signal (i.e. software interrupt) which will be triggered
  upon message reception. The signal service routine can
  then receive the incoming message and process it or notify
  the main program of the event in a user defined manner.
  This mechanism allows processes which are inherently compute
  bound to react quickly to incoming messages without using
  some form of message polling.

  A separate signal is used to indicate the arrival of
  emergency messages and trigger error recovery procedures.

  If a user's main program is performing a message wait
  of any form (see above) and any signal arrives, the
  message wait is terminated with an error.  This is
  the same as the effect of a signal on a UNIX Read call.
  The user making use of signals in a message program
  should be prepared to handle these aborted calls.

  DefineSignal(MsgType,Signal) is used to define which of
  the 16 UNIX signals a process wishes to use for handling
  messages of type msgtype (see below).  Two message types
  exist: (1) NORMALMSG and (2) EMERGENCYMSG.  Thus two
  separate handlers may be defined for the receipt of messages.
  By specifying a signal of zero, the signal handling of
  messages of that type is disabled. The binding of a
  routine to a signal specified by this call must be
  separately made using the normal UNIX call signal.
*/

GetSignal(MsgType)

/*
  Because UNIX signals are not stacked, various calls
  are provided to allow a process to correctly handle
  multiple message events. If a message handler is enabled
  and the last signalled message event has not yet been
  handled, a counter is incremented and no new signal is
  generated.  The state of this counter can be discovered
  using GetSignal.
*/

ClearSignal(MsgType)

/*
  The signal counter may be cleared by ClearSignal
  which also reenables further signals.
*/

DismissSignal(MsgType)

/*
  This routine should be called by the signal handling
  routine after each message is processed.  It decrements
  the counter and returns the total number
  of messages still waiting to be handled by that handler
  (message events could occur while the handler is running
  so this count may not necessarily decrease).
  A returned value of zero implies that the handler has
  no more work to do and that message events from that
  point on will cause a new signal to be generated (and
  thus a new incarnation of the handler to be called).
*/

MakeFileDescriptor(LocalPort,RemotePort)
  LocalPortName LocalPort;
  LocalPortName RemotePort;

/*
    Make a file descriptor corresponding to
    a pair of ports.
*/


LocalPortName LocalPort(FileDescriptor)

/*
    Decompose a file descriptor.
*/



LocalPortName RemotePort(FileDescriptor)

/*
    Decompose a file descriptor.
```

```
*/

SetBacklog(Port,NewBacklog)
  LocalPortName Port;
  int NewBacklog;

/*
    Change the backlog of a port.
*/
```

13. Acknowledgements

The author would like to acknowledge the help, advice and assistance of Eugene Ball, George Robertson, Peter Hibbard and Sam Harbison at CMU, Keith Lantz and Ed Burke at the University of Rochester, Greg Jirak, David Kashtan and other members of the MUSE project at SRI.

References

1. J.E. Ball, J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner., "RIG, Rochester's Intelligent Gateway: System overview.," *IEEE Trans. Software Eng.*, Vol. 2, No. 4, December 1976, pp. 321-328.

2. J. E. Ball, E. Burke, I. Gertner, K. A. Lantz, and R. F. Rashid, "Perspectives on message-based distributed computing," *Proceedings 1979 Networking Symposium*, IEEE, December 1979, pp. .

3. K.A. Lantz and R.F. Rashid, "Virtual terminal management in a multiple process environment," *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, December 1979, pp. 86-97.

4. D. J. Farber, et al., "The Distributed Computing System," *Proceedings of IEEE COMPCON*, IEEE, 1973, pp. 31-34.

5. D.L. Mills, "An overview of the Distributed Computer Network," *Proceedings AFIPS NCC*, AFIPS, June 1976, pp. 45:523-531.

6. NSW Protocol Committee, "MSG: The interprocess communication facility for the National Software Works," Tech. report 3483, Bolt Beranek and Newman, December 1976.

7. J.E. White, "A high-level framework for network-based resource sharing," *Proceedings AFIPS NCC*, AFIPS, June 1976, pp. 45:561-570.

8. R.W. Watson and J.G. Fletcher, "A protocol structure for network operating system services," *Proceedings 4th Berkeley Conference on Distributed Data Management*, August 1979, pp..

9. F. Baskett, J.H. Howard, and J.T. Montague, "Task communication in DEMOS," *Proceedings of the 6th Symposium on Operating Systems Principles*, November 1977, pp. 23-32.

10. M.H. Solomon and R.A. Finkel, "Roscoe -- A multiminicomputer operating system," Tech. report 321, Computer Science Department, University of Wisconsion-Madison, September 1978.

11. Cheriton, D.R.; Malcolm, M.A.; Melen, L.S.; and Sager, G.R., "Thoth, a Portable Real-Time Operating System," *CACM*, February 1979, pp. 105-115.

12. S. Ward, "TRIX: A network operating system," Tech. report, MIT, December 1979.

13. J.A. Feldman, "High-level programming for distributed computing," *CACM*, Vol. 22, No. 6, June 1979, pp. 353-368.

14. , "Proposal for a joint effort in personal scientific computing," Tech. report, Computer Science Department, Carnegie-Mellon University, August 1979.

15. B. Liskov, "Primitives for distributed computing," *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, December 1979, pp. 33-42.

Path of Message Communication

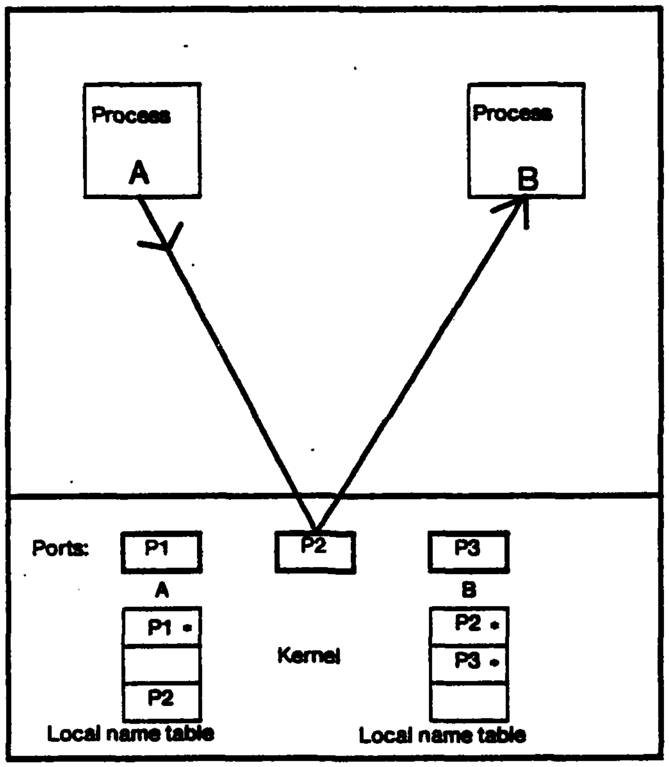


Figure 1

Starred port names imply ownership.

Simple Message Formats

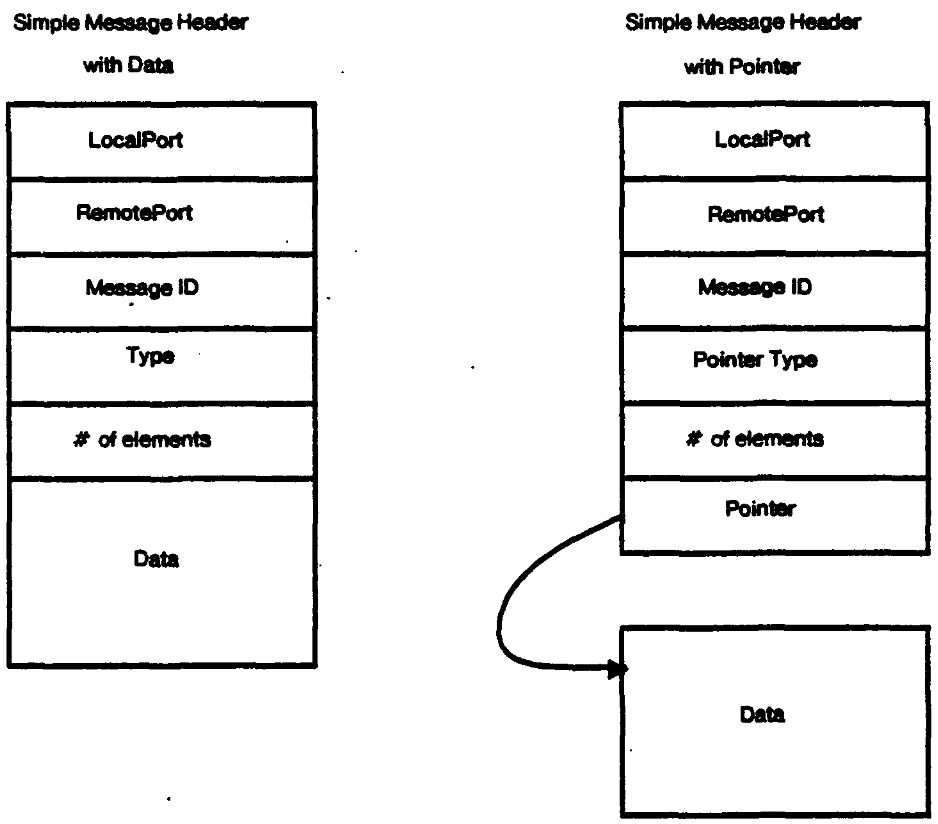


Figure 2

Message Data Descriptors

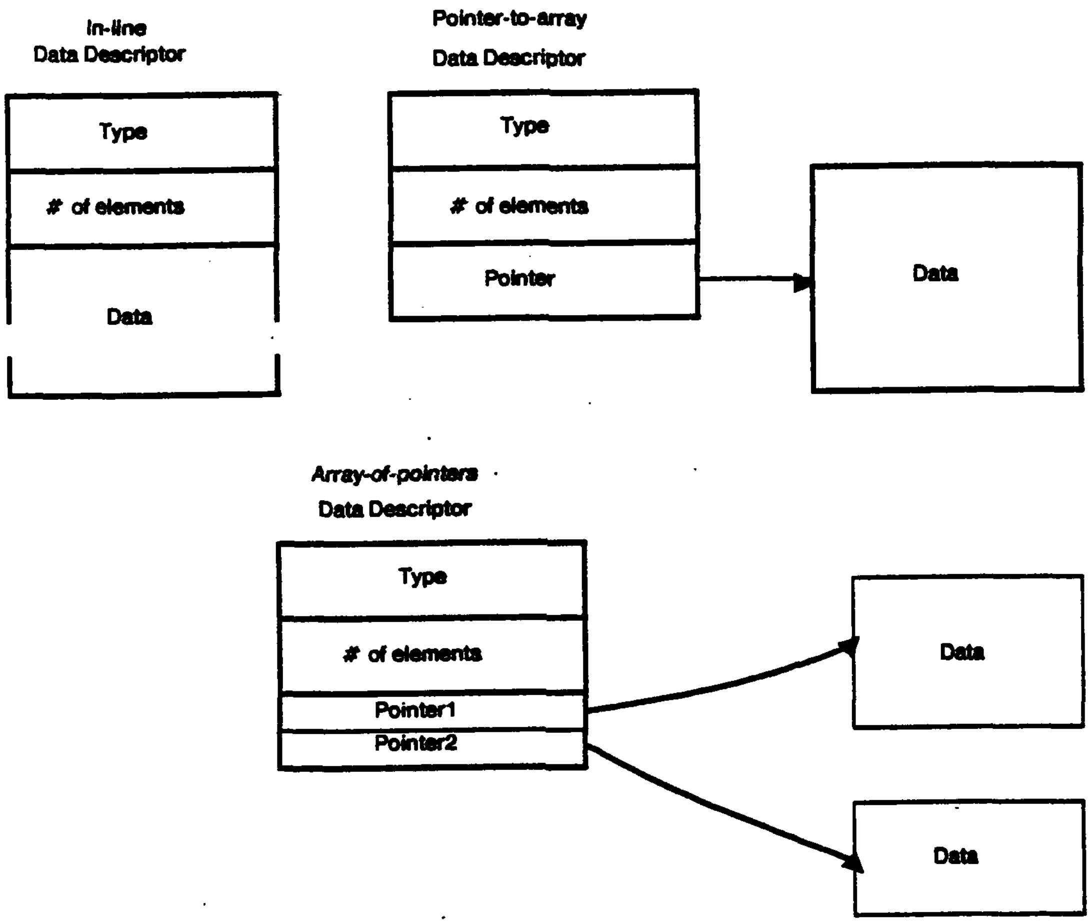


Figure 3

Structured Data Description

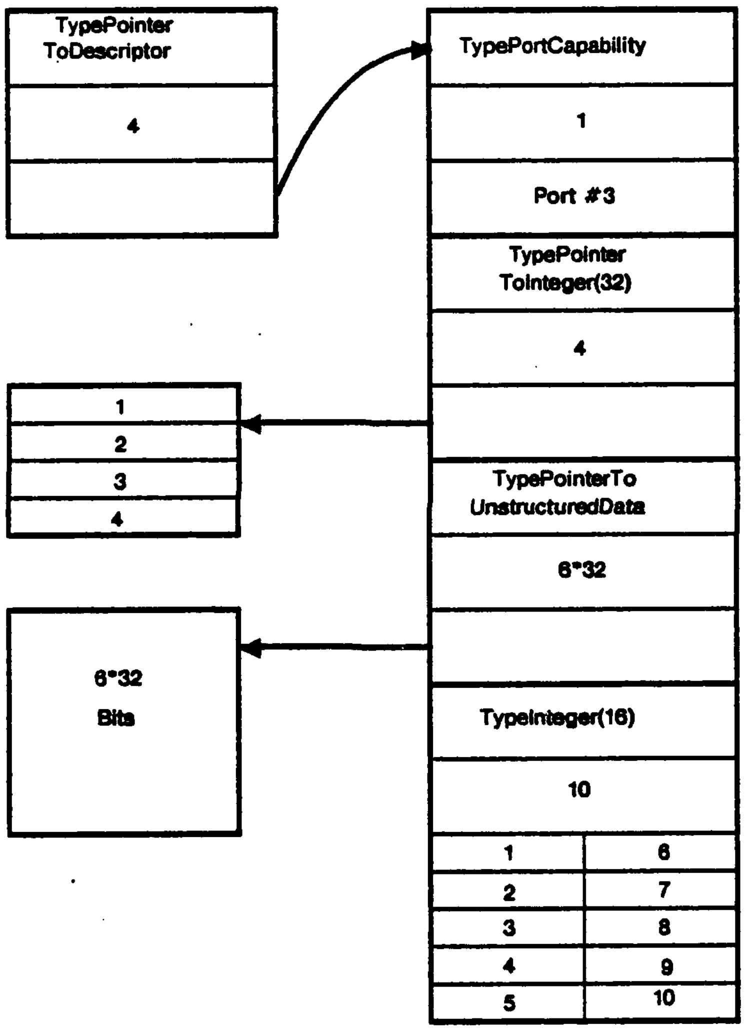
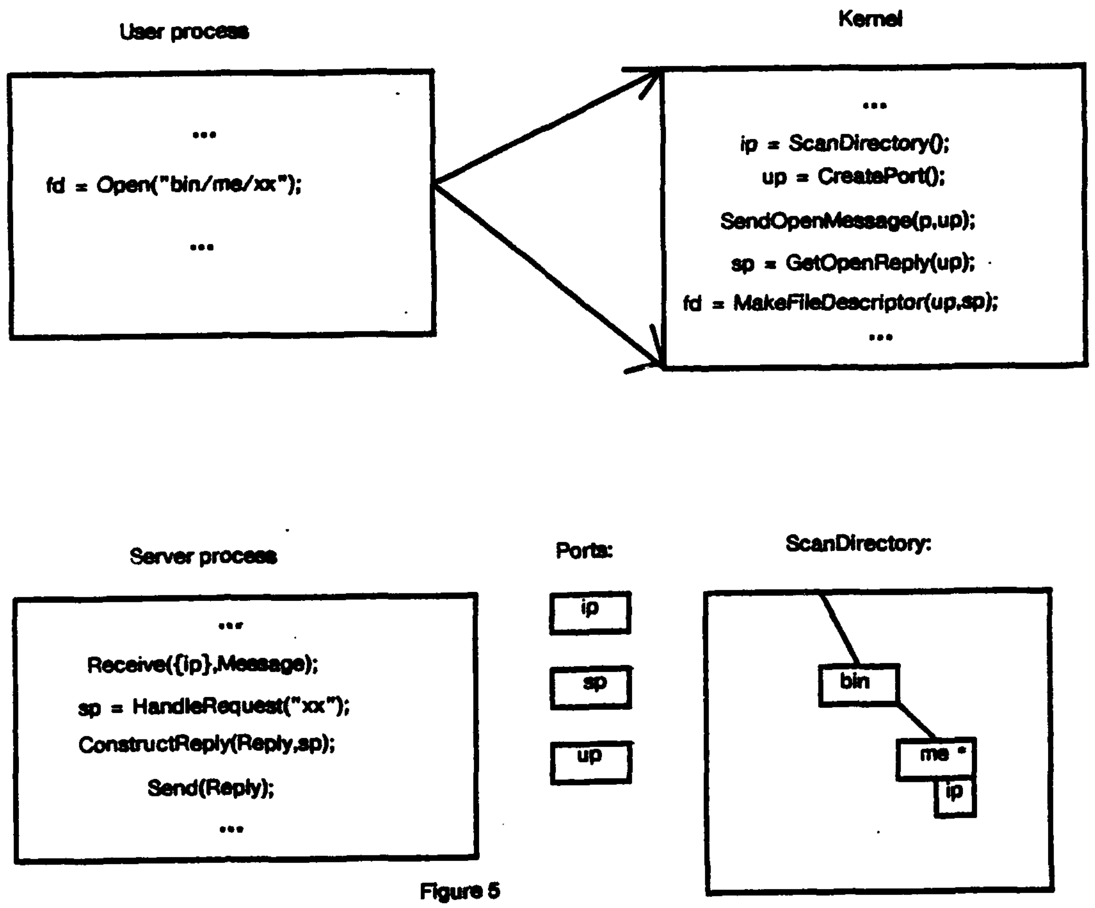


Figure 4

Processing an Open Request



```
...
fd = Open("bin/me/xx");
...
```

```
...
ip = ScanDirectory();
up = CreatePort();
SendOpenMessage(p,up);
sp = GetOpenReply(up);
fd = MakeFileDescriptor(up,sp);
...
```

```
...
Receive({ip},Message);
sp = HandleRequest("xx");
ConstructReply(Reply,sp);
Send(Reply);
...
```

Figure 5

* = special file
ip = initial connection port for server
sp = server port for handling requests on this connection
up = user port created for connection

Two Processes Communicating Via Network

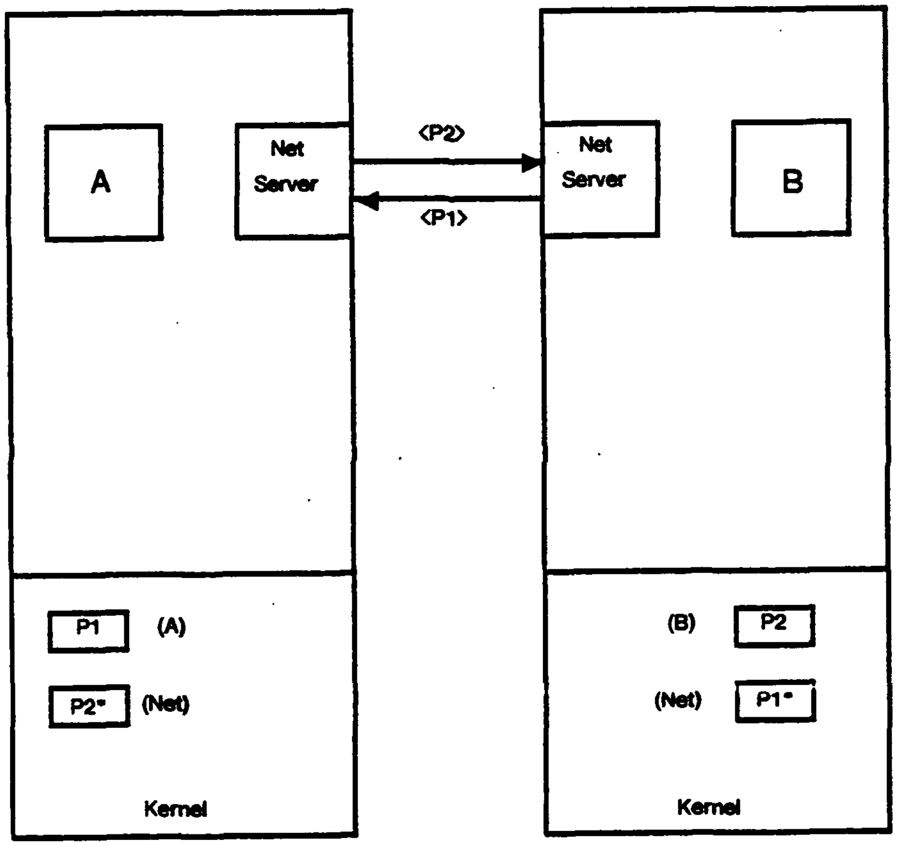


Figure 6

Starred ports are owned by network servers
and correspond to ports on another processor